```
val it = ( ) : unit
```

A session with ML

Points

(Although ML stands for Meta Language, we are dealing with a real and very elegant programming language!)

The presentation is a series of examples (all of them run in our Unix-based dialect of ML). There is much more to ML — as there is to Scheme — so this is only an attempt to whet your appetite ☺.

```
% sml
Standard ML of New Jersey,
Version 75, November 11, 1991
val it = ( ) : unit
```

This tells us how to interpret ML's output: `it` stands for the value of the expression evaluated in this step of the top-level loop; `unit` is the type. ML's prompt is the `-` sign, and it changes to `=` when the expression has several lines of parts. Terminate your input with a semicolon.

```
(* the simplest stuff *)
-
2 + 3 * 4;
val it = 14 : int
```

A function definition. Note that the value of `succ` is the functional expression `fn:int->int`.

```
-
fun succ x = x + 1;
val succ = fn : int -> int
```

And an application of this function:

```
-
3 * succ 4 * succ 5;
val it = 90 : int
```

ML is great with lists (as expected!).

```
-
fun length( x ) =
    if null( x ) then 0
    else 1+length( tl( x ) );
val length = fn : 'a list -> int
```

The type of list elements has not been determined. ML leaves it open, as indicated by the type "placeholder" `'a`: a list of "things" maps into `int`.

```
-
length( [11, 33, 55] );
val it = 3 : int
-
length( ["11", "abc"] );
val it = 2 : int
```


The same function could be defined as a series of patterns (rather like in Prolog):

```
-
fun
  length( nil ) = 0 |
  length( a::x ) = 1+length( x );
val length = fn : 'a list -> int
```

A small test:

```
-
length( nil );
val it = 0 : int
```

Yet another form, without parentheses:

```
-
fun
  length nil = 0 |
  length ( a::x ) = 1+length x;
val length = fn : 'a list -> int

-
length ( ["a", "bb", "ccc"] );
val it = 3 : int
```

(Note that the type of the elements is not important!)

No presentation would be complete without this:

```
-
fun append( x, z ) =
    if null( x ) then z else
    hd(x) :: append( tl(x), z );
val append = fn :
    'a list * 'a list -> 'a list
```

(The `::` denotes list construction, the same as `cons` in Scheme.) The arguments are lists of “things”, as is the value of the function. The `*` denotes the cross-product.

```
-
append([1, 2, 3, 4], [5, 6, 7]);
val it = [1,2,3,4,5,6,7] :
    int list
```

And a definition with patterns:

```
-
fun
  append( nil, z ) = z |
  append( a::y, z ) =
          a :: append( y, z );
val append = fn :
    'a list * 'a list -> 'a list
```


A simple application:

```
-
append( [1, 2, 3, 4], [5, 6] );
val it = [1,2,3,4,5,6] : int list
```

Another application?... Ouch!

```
-
append( ["a", "b"], [3] );
std_in:1.1-1.25 Error: operator and operand
don't agree (tycon mismatch)
  operator domain: string list * string list
  operand:         string list * int list
  in expression:
    append ("a" :: "b" :: nil,3 :: nil)
```

That’s right: ML requires type agreement! This will work—there are only strings on the lists.

```
-
append(["a", "b"], ["cc", "dd"]);
val it = ["a", "b", "cc", "dd"] :
    string list
```

Incidentally, `append` is built-in, naturally, and conveniently available as an infix operator:

```
-
["a", "b"] @ ["cc", "dd"];
val it = ["a", "b", "cc", "dd"] :
    string list
```

By the way, string concatenation is available too:

```
-
"abcd" ^ "efghijk";
val it = "abcdefghijk" : string
```

More function definitions... This reverses the first list and tucks it onto the second list:

```
-
fun reverse( nil, z ) = z |
    reverse( a::y, z ) =
             reverse( y, a::z );
val reverse = fn :
    'a list * 'a list -> 'a list
```

Will it work?...

```
-
reverse( [1, 2, 3], [4] );
val it = [3, 2, 1, 4] : int list
```

Whew. Now, how do we reverse a list?

```
-
fun rev x = reverse( x, nil );
val rev = fn : 'a list -> 'a list
```

Does this work?

```
-
rev( [1, 2, 3] );
val it = [3, 2, 1] : int list
```


We already did this in Scheme:

```
-
fun same_neighbours L =
   if null L then false else
   if null (tl L) then false else
   if hd L = hd (tl L) then true
   else same_neighbours (tl L);
val same_neighbours =
    fn : ''a list -> bool

-
same_neighbours [3, 4, 5, 6];
val it = false : bool
-
same_neighbours [3, 4, 4, 5, 6];
val it = true : bool
```

The same with patterns:

```
-
fun
 same_neighbours nil = false |
 same_neighbours (a::nil) =
      false |
 same_neighbours (a::b::L) =
   if a = b then
      true
   else
      same_neighbours (b::L);
```

Type inference in ML is very elaborate, and quite powerful. First, what happens when operand types are not specified? Here ML notices that `1` is an integer:

```
-
fun succ x = x + 1;
val succ = fn : int -> int
```

Here, `1.0` is a real number:

```
-
fun succr x = x + 1.0;
val succr = fn : real -> real
```

Here, however, there is nothing to help ML:

```
-
fun sq x = x * x;
std_in:5.13 Error: overloaded variable "*"
cannot be resolved
```

A hint is necessary—just one hint will be enough:

```
-
fun sq x: int = x * x;
val sq = fn : int -> int
```

Or any of these:

```
- fun sq x = x * x : int;
val sq = fn : int -> int

- fun sq x =(x: int) * x;
val sq = fn : int -> int
```


Precedence of operations in ML—one example...

```
-
length 7::[];
std_in:10.1-10.12 Error: operator and
operand don't agree ( tycon mismatch )
 operator domain: 'Z list
 operand:         int
 in expression:
   length 7
```

Try parentheses to evaluate `::` before `length`.

```
-
length( 7::[] );
val it = 1 : int
-
length( 7::8::nil );
val it = 2 : int
```

Two other (more elaborate) examples:

```
-
"a"::"bb"::nil @
"c"^"cc"::"dddd"::"eee"::nil;
val it =
  ["a","bb","ccc","dddd","eee"] :
  string list

-
length ["abcd"]::2*11::nil @
333::4400+44::555::nil;
val it = [1,22,333,4444,555] :
         int list
```

Higher-order functions are very similar to the same functions in Scheme. First, the built-in `map`.

```
-
map sq [1, 3, 5];
val it = [1, 9, 25] : int list
```

(Observe the parenthesis-free notation.)

```
-
map
  sq
  ( map hd [ 1::[11],
             2::[22, 222, 2222],
             3::[33, 333]] );
val it = [1, 4, 9] : int list
```

Here's how this form is defined:

```
-
fun map f nil = nil |
    map f ( a::y ) =
        ( f a ) :: map f y;
val map = fn :
    ( 'a -> 'b ) ->
    'a list ->
    'b list
```

The interpretation of this functional value is a little complicated: `map f` is a function from `'a list` to `'b list`.

Let's explore this situation on a simpler example:

```
- fun add x y: int = x + y;
val add = fn : int -> int -> int
```

Here, `add x` is a function from `int` to `int`. In particular, `add 2` is such a function:

```
- val succ2 = add 2;
val succ2 = fn : int -> int
- succ2 7;
val it = 9 : int
```

Similarly, `map sq` is a function from `int list` to `int list`, and `map length` is a function from a list of lists to a list of integers.

```
-
val squarelist = map sq;
val squarelist = fn :
    int list -> int list
-
squarelist [5,7,11];
val it = [25,49,121] : int list
-
val lengths = map length;
val lengths = fn :
    'a list list -> int list
-
lengths [[1], [2, 3], [4,5,6]];
val it = [1,2,3] : int list
```

A form of `map` with parentheses is also possible:

```
-
fun mapp( f, nil ) = nil |
    mapp( f, a::y ) =
       ( f a ) :: mapp( f, y );
val mapp = fn :
    ('a -> 'b) * 'a list ->
    'b list

-
mapp( sq, [1, 2, 3] );
val it = [1, 4, 9] : int list
```

`map` works well with anonymous functions (they correspond to lambda expressions in Scheme):

```
-
map ( fn x => x*x*x ) [2, 3, 4];
val it = [8, 27, 64] : int list

-
val sq = fn x:int => x*x;
val sq = fn : int -> int

- sq 12;
val it = 144 : int
```

By the way, to negate a number use ~:

```
- sq ~12;
val it = 144 : int
```


Function composition:

```
-
map ( sq o sq ) [2, 3, 4];
val it = [16, 81, 256] : int list

-
val pow4 = sq o sq;
val pow4 = fn : int -> int
- pow4 4;
val it = 256 : int

-
val second = hd o tl;
val second = fn : 'a list -> 'a
- second [5, 3, 8];
val it = 3 : int
```

Precedences may be confusing—write `(hd o tl)`.

```
- hd o tl [5, 3, 8];
std_in:2.1-2.17 Error: operator and operand don't agree
(tycon mismatch)
  operator domain: ('Z list -> 'Z) * ('Y -> 'Z list)
  operand:         ('Z list -> 'Z) * int list
  in expression:
    o (hd,tl 5 :: <exp> :: <exp>)
```

And now, the dessert ☺:

```
-
(hd o tl)
[fn x=>x:int, fn x=>x*x:int] 7;
val it = 49 : int
```

Reducers (see the corresponding Scheme notes!):

```
-
fun reduce(f, nil, v0) = v0 |
    reduce(f, ( a::y ), v0) =
      f(a, reduce(f, y, v0));
val reduce = fn : ('a * 'b -> 'b)
* 'a list * 'b -> 'b
```

We can use `reduce` with anonymous functions:

```
- reduce(fn(x, y:int)=>x+y,
         [1, 2, 3, 4], 0);
val it = 10 : int
- reduce(fn(x, y:int)=> x*y,
         [1, 2, 3, 4], 1);
val it = 24 : int
```

The same, more simply, with an operator promoted to a function:

```
- reduce(op +, [1, 2, 3, 4], 0);
val it = 10 : int
- reduce(op *, [1, 2, 3, 4], 1);
val it = 24 : int
```

Finally, a parenthesis-free version of `reduce`:

```
fun reduce f nil v0 = v0 |
    reduce f ( a::y ) v0 =
      f( a, reduce f y v0 );
val reduce = fn :
  ( 'a * 'b -> 'b ) -> 'a list -> 'b -> 'b
```


User-defined data types. An enumerated type:

```
-
datatype colour =
         red | amber | green;
datatype  colour
con amber : colour
con green : colour
con red : colour

- red;
val it = red : colour

- length [red,green,red,amber];
val it = 4 : int
```

A type with functions as members:

```
-
datatype tree =
 nul | node of int * tree * tree;
datatype  tree
con node :
    int * tree * tree -> tree
con nul : tree

- node;
val it = fn :
    int * tree * tree -> tree
```

ML checks completeness of definitions:

```
- fun left(node(a, L, R)) = L;
std_in:2.1-2.31 Warning: match not
exhaustive
        node (a,L,R) => ...
val left = fn : tree -> tree
```

We can use **exceptions** to make it complete:

```
- exception NoRight;
exception NoRight;
- fun right(node(a, L, R)) = R |
        right(nul) = raise NoRight;
val right = fn : tree -> tree
- right nul;
uncaught exception NoRight
```

Inserting into a tree (treated as a BST):

```
-
fun
insert( a, nul ) =
   node( a, nul, nul ) |
insert( a, node( b, L, R ) ) =
   if a < b then
     node( b, insert( a, L ), R )
   else if a > b then
     node( b, L, insert( a, R ) )
   else node( b, L, R );
val insert = fn :
    int * tree -> tree
```


Tree traversal:

```
- fun inorder(nul) = nil |
      inorder(node(a, L, R)) =
    inorder(L) @ (a::inorder(R));
val inorder = fn :
    tree -> int list

-
val my_tree =
    insert(7, insert(3, insert(9,
    insert(4, insert(3, nul)))));
val my_tree = node (3,nul,
    node (4,nul,node #)) : tree

- right(my_tree);
val it = node (4,nul,
         node (9,node #,nul)) :
                            tree

- right(right(my_tree));
val it = node (9,
         node (7,nul,nul),nul) :
                             tree

-
inorder( my_tree );
val it = [3,4,7,9] : int list
```

Locality in ML.

```
-
let val aa = [1,2]
in tl aa
end;
val it = [2] : int list
```

But aa remains undefined:

```
-
aa;
std_in:2.1-2.2 Error: unbound
variable or constructor aa
```

More local objects:

```
-
let val aa=[1,2] and bb=[3,4,5]
in aa @ bb
end;
val it = [1,2,3,4,5] : int list
```

Nesting is also possible

```
-
let val aa = [1,2]
in let val bb = [3,4,5]
   in aa @ bb
   end
end;
val it = [1,2,3,4,5] : int list
```


The same without nesting:

```
-
let val (aa,    bb) =
        ([1,2], [3,4,5])
in aa @ bb
end;
val it = [1,2,3,4,5] : int list
```

Local functions:

```
-
local fun divides(x, y) =
          y mod x = 0
in fun anniversary age =
          divides(10, age) orelse
          divides(25, age)
end;
val anniversary = fn :
    int -> bool

- anniversary 30;
val it = true : bool

- anniversary 45;
val it = false : bool
```

Local patterns:

```
-
fun mirror ( p as ( x, y ) ) =
   ( p, ( y, x ) );
val mirror = fn :
 'a * 'b -> ('a * 'b) * ('b * 'a)

-
mirror (6,17);
val it = ((6,17),(17,6)) :
    (int * int) * (int * int)
```

Minimum of a list:

```
-
local
  fun minl_aux(elt, lst): int =
    if null lst then elt
    else if elt > hd(lst) then
       minl_aux(hd lst, tl lst)
    else minl_aux(elt, tl lst)
in
  fun minl L =
    if null L then ~1000000000
    else minl_aux(hd L, tl L)
end;
val minl = fn : int list -> int

- minl [1, 2, 3, 0, 5, 4, ~9, 8];
val it = ~9 : int
```


Tuples versus records in ML.

```
-
( 3, 4 ) = ( 4, 3 );
val it = false : bool
```

Records have named **fields**, and the order of the field values in a record is not important:

```
-
{ a=3, b=4 } = { b=4, a=3 };
val it = true : bool
```

Tuples have elements of the same type:

```
-
( 3, "four" ) = ( "four", 3 );
std_in:3.1-3.29 Error: operator and
operand don't agree (tycon mismatch)
  operator domain: (int * string) * (int *
string)
  operand:         (int * string) *
(string * int)
  in expression:
    = ((3,"four"),("four",3))
```

Records may have elements of mixed types:

```
-
{ a = 3, b = "four" } =
{ b = "four", a = 3 };
val it = true : bool
```

Generic types in ML.

```
- datatype 'a list =
  null | cons of 'a * ('a list);
datatype 'a  list
con cons :
 'a * 'a list -> 'a list
con null : 'a list

- cons(1, cons(2, null));
val it = cons (1,cons (2,null)) :
    int list
- cons("aa", cons("bb", null));
val it =
  cons ("aa",cons ("bb",null)) :
  string list

- val twoFunc =
        cons(fn x:int=>x*x*x,
          cons(fn x=>x*x, null));
val twoFunc = cons (fn,cons
(fn,null)) : (int -> int) list

- fun head(cons(x, y)) = x;
std_in:8.1-8.27 Warning: match not
exhaustive
        cons (x,y) => ...
val head = fn : 'a list -> 'a

- head(twoFunc) 3;
val it = 27 : int
```


ML——summary

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....